Towards Efficient Abstractions for Concurrent Consensus*

Carlo Spaccasassi** and Vasileios Koutavas***

Trinity College Dublin, Ireland
{spaccasc,Vasileios.Koutavas}@scss.tcd.ie

Abstract. Consensus is an often occurring problem in concurrent and distributed programming. We present a programming language with simple semantics and build-in support for consensus in the form of communicating transactions. We motivate the need for such a construct with a characteristic example of generalized consensus which can be naturally encoded in our language. We then focus on the challenges in achieving an implementation that can efficiently run such programs. We setup an architecture to evaluate different implementation alternatives and use it to experimentally evaluate runtime heuristics. This is the basis for a research project on realistic programming language support for consensus.

Keywords: Concurrent programming, consensus, communicating transactions

1 Introduction

Achieving consensus between concurrent processes is a ubiquitous problem in multicore and distributed programming [8, 6]. Among the classic instances of consensus is leader election and synchronous multi-process communication. Programming language support for consensus, however, has been limited. For example, CML's first-class communication primitives provide a programming language abstraction to implement two-party consensus. However, they cannot be used to abstractly implement consensus between three or more processes [11, Thm. 6.1]—this needs to be implemented in a case-by-case basis.

Let us consider a hypothetical scenario of generalized consensus, which we will call the *Saturday Night Out* (SNO) problem. In this scenario a number of friends are seeking partners for various activities on Saturday night. Each has a list of desired activities to attend in a certain order, and will only agree for a night out if there is a partner for each activity. Alice, for example, is looking for company to go out for dinner and then a movie (not necessarily with the same person). To find partners for these events in this order she may attempt to synchronize on the "handshake" channels `dinner` and `movie`:

* Student project paper (primarily the work of the first author).

** Supported by MSR (MRL 2011-039)

*** Supported by SFI project SFI 06 IN.1 1898.

$\mathsf{Alice} \stackrel{\text{def}}{=}$ **sync** `dinner`; **sync** `movie`

Here **sync** is a two-party synchronization operator, similar to CSP synchronization. Bob, on the other hand, wants to go for dinner and then for dancing:

$\mathsf{Bob} \stackrel{\text{def}}{=}$ **sync** `dinner`; **sync** `dancing`

Alice and Bob can agree on dinner but they need partners for a movie and dancing, respectively, to commit to the night out. Their agreement is *tentative*.

Let Carol be another friend in this group who is only interested in dancing:

$\mathsf{Carol} \stackrel{\text{def}}{=}$ **sync** `dancing`

Once Bob and Carol agree on dancing they are both happy to commit to going out. However, Alice has no movie partner and she can still cancel her agreement with Bob. If this happens, Bob and Carol need to be notified to cancel their agreement and everyone starts over their search of partners. An implementation of the SNO scenario between concurrent processes would need to have a specialized way of reversing the effect of this synchronization. Suppose David is also a participant in this set of friends.

$\mathsf{David} \stackrel{\text{def}}{=}$ **sync** `dancing`; **sync** `movie`

After the partial agreement between Alice, Bob, and Carol is canceled, David together with the first two can synchronize on `dinner`, `dancing`, and `movie` and agree to go out (leaving Carol at home).

Notice that when Alice raised an objection to the agreement that was forming between her, Bob, and Carol, all three participants were forced to restart. If, however, Carol was taken out of the agreement (even after she and Bob were happy to commit their plans), David would have been able to take Carol's place and the work of Alice and Bob until the point when Carol joined in would not need to be repeated.

Programming SNO between an arbitrary number of processes (which can form multiple agreement groups) in CML is complicated. Especially if we consider that the participants are allowed to perform arbitrary computations between synchronizations affecting control flow, and can communicate with other parties not directly involved in the SNO. For example, Bob may want to go dancing only if he can agree with the babysitter to stay late:

$\mathsf{Bob} \stackrel{\text{def}}{=}$ **sync** `dinner`; **if** `babysitter()` **then** **sync** `dancing`

In this case Bob's computation has side-effects outside of the SNO group of processes. To implement this would require code for dealing with the SNO protocol to be written in the Babysitter (or any other) process, breaking any potential modular implementation.

This paper shows that *communicating transactions*, a recently proposed mechanism for automatic error recovery in CCS processes [13], is a useful mechanism for modularly implementing the SNO and other generalized consensus scenarios. Previous work on communicating transactions focused on behavioral theory with

$$
\begin{array}{rl}
A ::= & \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{int} \mid A \times A \mid A \to A \mid A\,\mathbf{chan} \\
v ::= & x \mid () \mid \mathbf{true} \mid \mathbf{false} \mid n \mid (v, v) \mid \mathbf{fun}\, f(x) = e \mid c \\
e ::= & v \mid (e, e) \mid e\ e \mid op\ e \mid \mathbf{let}\, x = e\, \mathbf{in}\, e \mid \mathbf{if}\, e\, \mathbf{then}\, e\, \mathbf{else}\, e \\
\mid & \mathbf{send}\, e\ e \mid \mathbf{recv}\, e \mid \mathbf{newChan}_A \mid \mathbf{spawn}\, e \\
\mid & \mathbf{atomic}\, [\![\, e \rhd_k e \,]\!] \mid \mathbf{commit}\, k \\
P ::= & e \mid P \parallel P \mid \nu c.P \mid [\![\, P \rhd_k P \,]\!] \mid \mathbf{co}\, k \\
op ::= & \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{leq} \\
E ::= & [] \mid (E, e) \mid (v, E) \mid E\ e \mid v\ E \mid op\ E \mid \mathbf{let}\, x = E\, \mathbf{in}\, e \\
\mid & \mathbf{if}\, E\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid \mathbf{send}\, E\ e \mid \mathbf{send}\, v\ E \mid \mathbf{recv}\, E \mid \mathbf{spawn}\, E
\end{array}
$$

where $n \in \mathbb{N}$, $x \in \mathit{Var}$, $c \in \mathit{Chan}$, $k \in \mathcal{K}$

Fig. 1. TCML syntax.

respect to *safety* and *liveness* [13, 2]. However, the effectiveness of this construct in a pragmatic programming language has yet to be proven. One of the main milestones to achieve on this direction is the invention of efficient runtime implementations of communicating transactions. Here we describe the challenges and our first results in a recently started project to investigate this research direction.

In particular, we equip a simple concurrent functional language with communicating transactions and use it to discuss the challenges in making an efficient implementation of such languages (Sect. 2). We also use this language to give a modular implementation of consensus scenarios such as the SNO example. The simple operational semantics of this language allows for the communication of SNO processes with arbitrary other processes (such as the Babysitter process) without the need to add code for the SNO protocol in those processes. Moreover, the more efficient, partially aborting strategy discussed above is captured in this semantics.

Our semantics of this language is non-deterministic, allowing different runtime scheduling strategies of processes, some more efficient than others. To study their relative efficiency we have developed a skeleton implementation of the language which allows us to plug in and evaluate such runtime strategies (Sect. 3). We describe several such strategies (Sect. 4) and report the results of our evaluations (Sect. 5). Finally, we summarize related work in this area and the future directions of this project (Sect. 6).

2 The TCML Language

We study TCML, a language combining a simply-typed λ-calculus with π-calculus and communicating transactions. For this language we use the abstract syntax shown in Fig. 1 and the usual abbreviations from the λ- and π-calculus. Values in TCML are either constants of base type (**unit**, **bool**, and **int**), pairs of values (of type $A \times A$), recursive functions ($A \to A$), and channels carrying values of type A ($A\,$**chan**). A simple type system (with appropriate progress

If-True	$\textbf{if}\,\textbf{true}\,\textbf{then}\,e_1\,\textbf{else}\,e_2$	$\hookrightarrow$	e_1	
If-False	$\textbf{if}\,\textbf{false}\,\textbf{then}\,e_1\,\textbf{else}\,e_2$	$\hookrightarrow$	e_2	
Let	$\textbf{let}\,x = v\,\textbf{in}\,e$	$\hookrightarrow$	$e[v/x]$	
Op	$op\,v$	$\hookrightarrow$	$\delta(op, v)$	
App	$\textbf{fun}\,f(x) = e\ v_2$	$\hookrightarrow$	$e[\textbf{fun}\,f(x) = e/f][v_2/x]$	
Step	$E[e]$	$\longrightarrow$	$E[e']$	if $e \hookrightarrow e'$
Spawn	$E[\textbf{spawn}\,v]$	$\longrightarrow$	$v\ () \parallel E[()]$	
NewChan	$E[\textbf{newChan}_A]$	$\longrightarrow$	$\nu c.E[c]$	if $c \notin \mathrm{fc}(E[()])$
Atomic	$E[\textbf{atomic}\,[\![\, e_1 \rhd_k e_2 \,]\!]]$	$\longrightarrow$	$[\![\, E[e_1] \rhd_k E[e_2] \,]\!]$	
Commit	$E[\textbf{commit}\,k]$	$\longrightarrow$	$\textbf{co}\,k \parallel E[()]$	

Fig. 2. Sequential reductions

and preservation theorems) can be found in an accompanying technical report [12] and is omitted here.

Source TCML programs are expressions in the functional core of the language, ranged over by e, whereas running programs are processes derived from the syntax of P. Besides standard lambda calculus expressions, the functional core contains the constructs $\textbf{send}\,c\ e$ and $\textbf{recv}\,c$ to synchronously send and receive a value on channel c, respectively, and $\textbf{newChan}_A$ to create a new channel of type $\textbf{chan}\,A$. The constructs **spawn** and **atomic**, when executed, respectively spawn a new process and transaction; $\textbf{commit}\,k$ commits transaction k—we will shortly describe these constructs in detail.

A simple running process can be just an expression e. It can also be constructed by the parallel composition of P and Q ($P \parallel Q$). We treat free channels as in the π-calculus, considering them to be *global*. Thus if a channel c is free in both P and Q, it can be used for communication between these processes. The construct $\nu c.P$ encodes π-calculus restriction of the scope of c to process P. We use the Barendregt convention for bound variables and channels and identify terms up to alpha conversion. Moreover, we write $\mathrm{fc}(P)$ for the free channels in process P.

We write $[\![\, P_1 \rhd_k P_2 \,]\!]$ for the process encoding a communicating transaction. This can be thought of as the process P_1, the *default* of the transaction, which runs until the transaction *commits*. If, however, the transaction *aborts* then P_1 is discarded and the entire transaction is replaced by its *alternative* process P_2. Intuitively, P_2 is the continuation of the transaction in the case of an abort. As we will explain, commits are asynchronous, requiring the addition of process $\textbf{co}\,k$ to the language. The name k of the transaction is bound in P_1. Thus only the default of the transaction can potentially spawn a $\textbf{co}\,k$. The meta-function $\mathrm{ftn}(P)$ gives us the free transaction names in P.

Processes with no free variables can reduce using transitions of the form $P \longrightarrow Q$. These transitions for the functional part of the language are shown in Fig. 2 and are defined in terms of reductions $e \hookrightarrow e'$ (where e is a *redex*) and eager, left-to-right evaluation contexts E whose grammar is given in Fig. 1. Due to a unique decomposition lemma, an expression e can be decomposed to an evaluation context and a redex expression in only one way. Here we use $e[u/x]$

for the standard capture-avoiding substitution, and $\delta(op, v)$ for a meta-function returning the result of the operator op on v, when this is defined.

Rule Step lifts functional reductions to process reductions. The rest of the reduction rules of Fig. 2 deal with the concurrent and transactional side-effects of expressions. Rule Spawn reduces a $\mathbf{spawn}\, v$ expression at evaluation position to the unit value, creating a new process running the application $v\,()$. The type system of the language guarantees that value v here is a thunk. With this rule we can derive the reductions:

$$\begin{aligned}\mathbf{spawn}(\lambda().\,\mathbf{send}\, c\ 1); \mathbf{recv}\, c &\longrightarrow (\lambda().\,\mathbf{send}\, c\ 1)\ ()\ \|\ \mathbf{recv}\, c\\ &\longrightarrow \mathbf{send}\, c\ 1\ \|\ \mathbf{recv}\, c\end{aligned}$$

The resulting processes of these reductions can then communicate on channel c. As we previously mentioned, the free channel c can also be used to communicate with any other parallel process. Rule NewChan gives processes the ability to create new, locally scoped channels. Thus, the following expression will result in an input and an output process that can *only* communicate with each other:

$$\begin{aligned}&\mathbf{let}\, x = \mathbf{newChan_{int}}\ \mathbf{in}\, (\mathbf{spawn}\, (\lambda().\,\mathbf{send}\, x\ 1); \mathbf{recv}\, x)\\ &\longrightarrow \nu c.\, (\mathbf{spawn}\, (\lambda().\,\mathbf{send}\, c\ 1); \mathbf{recv}\, c)\\ &\longrightarrow^* \nu c.\, (\mathbf{send}\, c\ 1\ \|\ \mathbf{recv}\, c)\end{aligned}$$

Rule Atomic starts a new transaction in the current (expression-only) process, engulfing the entire process in it, and storing the abort continuation in the alternative of the transaction. Rule Commit spawns an asynchronous commit. Transactions can be arbitrarily nested, thus we can write:

$$\begin{aligned}&\mathbf{atomic}\, [\![\, \mathbf{spawn}(\lambda().\,\mathbf{recv}\, c; \mathbf{commit}\, k) \rhd_k ()\,]\!];\\ &\mathbf{atomic}\, [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!]\end{aligned}$$

which reduces to

$$\begin{aligned}&[\![(\mathbf{recv}\, c; \mathbf{commit}\, k)\ \|\ [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!]\\ &\rhd_k (); \mathbf{atomic}\, [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!]\,]\!]\end{aligned}$$

This process will commit the k-transaction after an input on channel c and the inner l-transaction after an input on d. As we will see, if the k transaction aborts then the inner l-transaction will be discarded (even if it has performed the input on d) and the resulting process (the alternative of k) will restart l:

$$(); \mathbf{atomic}\, [\![\, \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()\,]\!]$$

The effect of this abort will be the rollback of the communication on d reverting the program to a consistent state.

Process and transactional reductions are handled by the rules of Fig. 3. The first four rules (Sync, Eq, Par, and Chan) are direct adaptations of the reduction rules of the π-calculus, which allow parallel processes to communicate, and propagate reductions over parallel and restriction. These rules use an omitted

$$\textsc{Sync}\quad \frac{}{E_1[\mathbf{recv}\,c] \parallel E_2[\mathbf{send}\,c\,v] \longrightarrow E_1[v] \parallel E_2[()]}$$

$$\textsc{Eq}\quad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

$$\textsc{Par}\quad \frac{P_1 \longrightarrow P_1'}{P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2}$$

$$\textsc{Chan}\quad \frac{P \longrightarrow P'}{\nu c.P \longrightarrow \nu c.P'}$$

$$\textsc{Emb}\quad \frac{}{P_1 \parallel [\![\, P_2 \rhd_k P_3 \,]\!] \longrightarrow [\![\, P_1 \parallel P_2 \rhd_k P_1 \parallel P_3 \,]\!]}$$

$$\textsc{Step}\quad \frac{P \longrightarrow P'}{[\![\, P \rhd_k P_2 \,]\!] \longrightarrow [\![\, P' \rhd_k P_2 \,]\!]}$$

$$\textsc{Co}\quad \frac{P_1 \equiv \mathbf{co}\,k \parallel P_1'}{[\![\, P_1 \rhd_k P_2 \,]\!] \longrightarrow P_1'/k}$$

$$\textsc{Abort}\quad \frac{}{[\![\, P_1 \rhd_k P_2 \,]\!] \longrightarrow P_2}$$

Fig. 3. Concurrent and Transactional reductions (omitting symmetric rules).

structural equivalence ($\equiv$) to identify terms up to the reordering of parallel processes and the extrusion of the scope of restricted channels, in the spirit of the π-calculus semantics. Rule Step propagates reductions of default processes over their respective transactions. The remaining rules are taken from TransCCS [13].

Rule Emb encodes the *embedding* of a process P_1 in a parallel transaction $[\![\, P_1 \rhd_k P_2 \,]\!]$. This enables the communication of P_1 with P_2, the default of k. It also keeps the current continuation of P_1 in the alternative of k in case the k-transaction aborts. To illustrate the mechanics of the embed rule, let us consider the above nested transaction running in parallel with the process $P = \mathbf{send}\,d\,(); \mathbf{send}\,c\,()$:

$$\begin{array}{l} [\![(\mathbf{recv}\,c; \mathbf{commit}\,k) \parallel [\![\, \mathbf{recv}\,d; \mathbf{commit}\,l \rhd_l () \,]\!] \\ \rhd_k (); \mathbf{atomic}\,[\![\, \mathbf{recv}\,d; \mathbf{commit}\,l \rhd_l () \,]\!] \,]\!] \qquad \parallel \quad P \end{array}$$

After two embedding transitions we will have

$$[\![(\mathbf{recv}\,c; \mathbf{commit}\,k) \quad \parallel \quad [\![\, P \parallel \mathbf{recv}\,d; \mathbf{commit}\,l \;\; \rhd_l \;\; P \parallel () \,]\!] \quad \rhd_k \quad P \parallel \ldots]\!]$$

Now P can communicate on d with the inner transaction:

$$[\![(\mathbf{recv}\,c; \mathbf{commit}\,k) \quad \parallel \quad [\![\, \mathbf{send}\,c\,() \parallel \mathbf{commit}\,l \;\; \rhd_l \;\; P \parallel () \,]\!] \quad \rhd_k \quad P \parallel \ldots]\!]$$

Next, there are (at least) two options: either $\mathbf{commit}\,l$ spawns a $\mathbf{co}\,l$ process which causes the commit of the l-transaction, or the input on d is embedded in the l-transaction. Let us assume that the latter occurs:

$$\begin{array}{l} [\![\;\; [\![(\mathbf{recv}\,c; \mathbf{commit}\,k) \;\parallel\; \mathbf{send}\,c\,() \;\parallel\; \mathbf{commit}\,l \\ \quad \rhd_l (\mathbf{recv}\,c; \mathbf{commit}\,k) \;\parallel\; P \;\parallel\; () \,]\!] \\ \rhd_k P \parallel \ldots]\!] \\ \longrightarrow^* [\![\, [\![\, \mathbf{co}\,k \parallel \mathbf{co}\,l \rhd_l \ldots]\!] \rhd_k \ldots]\!] \end{array}$$

The transactions are now ready to commit from the inner-most to the outer-most using rule Commit. Inner-to-outer commits are necessary to guarantee that all transactions that have communicated have reached an agreement to commit.

This also has the important consequence of making the following three processes behaviorally indistinguishable:

$$[\![\, P_1 \rhd_k P_2 \,]\!] \parallel [\![\, Q_1 \rhd_l Q_2 \,]\!]$$

$$[\![\, P_1 \parallel [\![\, Q_1 \rhd_l Q_2 \,]\!] \rhd_k P_2 \parallel [\![\, Q_1 \rhd_l Q_2 \,]\!] \,]\!]$$

$$[\![\, [\![\, P_1 \rhd_k P_2 \,]\!] \parallel Q_1 \rhd_l [\![\, P_1 \rhd_k P_2 \,]\!] \parallel Q_2 \,]\!]$$

Therefore, an implementation of TCML, when dealing with the first of the three processes can pick any of the alternative, non-deterministic mutual embeddings of the k and l transactions without affecting the observable outcomes of the program. In fact, when one of the transactions has no possibility of committing or when the two transactions never communicate, an implementation can decide *never* to embed the two transactions in each-other. This is crucial in creating implementations that will only embed processes (and other transactions) only when necessary for communication, and pick the most *efficient* of the available embeddings. The development of implementations with efficient embedding strategies is one of the main challenges of our project for scaling communicating transactions to pragmatic programming languages.

Similarly, aborts are entirely non-deterministic (Abort) and are left to the discretion of the underlying implementation. Thus in the above example any transaction can abort at any stage, discarding part of the computation. In such examples there is usually a multitude of transactions that can be aborted, and in cases where a “forward” reduction is not possible (due to deadlock) aborts are necessary. Making the TCML programmer in charge of aborts (as we do with commits) is not desirable since the purpose of communicating transactions is to lift the burden of manual error prediction and handling. Minimizing aborts, and automatically picking the aborts that will undo the fewer computation steps while still rewinding the program back enough to reach a successful outcome is another major challenge in our project.

The SNO scenario can be simply implemented in TCML using *restarting transactions*. A restarting transaction uses recursion to re-initiate an identical transaction in the case of an abort:

$$\mathbf{atomic}_{rec\ k}\,[\![\, e \,]\!] \quad \stackrel{\text{def}}{=} \quad \mathbf{fun}\, r() = \mathbf{atomic}\,[\![\, e \rhd_k r\ () \,]\!]$$

A transactional implementation of the SNO participants we discussed in the introduction simply wraps their code in restating transactions:

$$
\begin{array}{ll}
\mathbf{let}\ \mathit{alice} & = \mathbf{atomic}_{rec\ k}\,[\![\, \mathit{sync\ dinner};\ \mathit{sync\ movie};\ \mathbf{commit}\, k \,]\!]\ \mathbf{in} \\
\mathbf{let}\ \mathit{bob} & = \mathbf{atomic}_{rec\ k}\,[\![\, \mathit{sync\ dinner};\ \mathit{sync\ dancing};\ \mathbf{commit}\, k \,]\!]\ \mathbf{in} \\
\mathbf{let}\ \mathit{carol} & = \mathbf{atomic}_{rec\ k}\,[\![\, \mathit{sync\ dancing};\ \mathbf{commit}\, k \,]\!]\ \mathbf{in} \\
\mathbf{let}\ \mathit{david} & = \mathbf{atomic}_{rec\ k}\,[\![\, \mathit{sync\ dancing};\ \mathit{sync\ movie};\ \mathbf{commit}\, k \,]\!]\ \mathbf{in} \\
\multicolumn{2}{l}{\mathbf{spawn}\ \mathit{alice};\ \mathbf{spawn}\ \mathit{bob};\ \mathbf{spawn}\ \mathit{carol};\ \mathbf{spawn}\ \mathit{david}}
\end{array}
$$

Here *dinner*, *dancing*, and *movie* are implementations of CSP synchronization channels and *sync* a function to synchronize on these channels. Compared

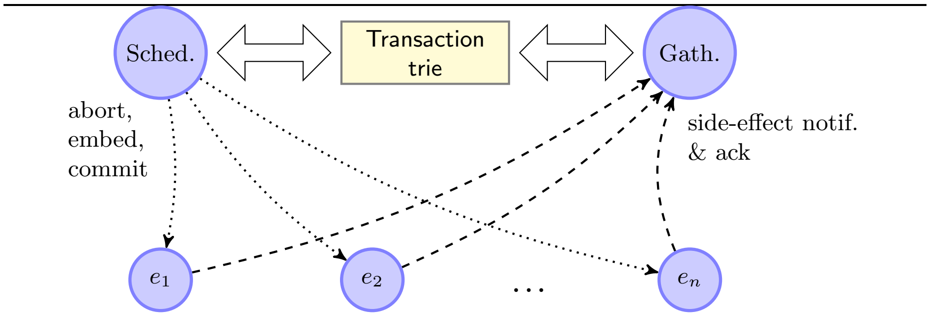

Fig. 4. TCML runtime architecture.

to a potential ad-hoc implementation of SNO in CML the simplicity of the above code is evident (the version of Bob communicating with the Babysitter is just as simple). However, as we discuss in Sect. 5, this simplicity comes with a severe performance penalty, at least for straightforward implementations of TCML. In essence, the above code asks from the underlying transactional implementation to solve an NP-complete satisfiability problem. Leveraging existing useful heuristics for such problems is something we intend to pursue in future work.

In the following sections we describe an implementation where these transactional scheduling decisions can be plugged in, and a number of heuristic transactional schedulers we have developed and evaluated. Our work shows that although more advanced heuristics bring measurable performance benefits, the exponential number of runtime choices require the development of innovative compilation and execution techniques to make communicating transactions a realistic solution for programmers.

3 An Extensible Implementation Architecture

We have developed an interpreter for the TCML reduction semantics in Concurrent Haskell [7, 10] to which we can plug-in different decisions about the non-deterministic transitions of our semantics. Here we briefly explain the runtime architecture of this interpreter, shown in Fig. 4.

The main Haskell threads are shown as round nodes in the figure. Each concurrent functional expression e_i is interpreted in its own thread according to the sequential reduction rules in Fig. 2 of the previous section. Side-effects in an expression will be generally handled by the interpreting thread, creating new channels, spawning new threads, and starting new transactions.

Except for new channel creation, the evaluation of all other side-effects in an expression will cause a *notification* (shown as dashed arrows in Fig. 2) to be sent to the *gatherer* process (Gath.). This process is responsible for maintaining a global view of the state of the running program in a *Trie* data-structure. This data-structure essentially represents the transactional structure of the program; i.e., the logical nesting of transactions and processes inside running transactions:

```
data TTrie = TTrie {
```

```
  threads   :: Set ThreadID,
  children  :: Map TransactionID TTrie, ... }
```

A `TTrie` node represents a transaction, or the top-level of the program. The main information stored in such a node is the set of threads (`threads`) and transactions (`children`) running in that transactional level. Each child transaction has its own associated `TTrie` node. An invariant of the data-structure is that each thread and transaction identifier appears only once in it. For example the complex program we saw on page 6:

$$\begin{array}{ll}\llbracket (\mathbf{recv}\, c; \mathbf{commit}\, k)^{\mathtt{tid}_1} \parallel \llbracket (\mathbf{recv}\, d; \mathbf{commit}\, l)^{\mathtt{tid}_2} \rhd_l () \rrbracket & \\ \rhd_k (); \mathbf{atomic} \llbracket \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l () \rrbracket \rrbracket & \parallel P^{\mathtt{tid}_P}\end{array}$$

will have an associated trie:

```
TTrie{threads  = {tid_P},
      children = {k ↦ TTrie{threads  = {tid_1},
                            children = {l ↦ TTrie{threads  = {tid_2},
                                                  children = ∅}}}}}
```

The last ingredient of the runtime implementation is the *scheduler* thread (Sched. in Fig. 4). This makes decisions about the commit, embed and abort transitions to be performed by the expression threads, based on the information in the trie. Once such a decision is made by the scheduler, appropriate signals (implemented using Haskell asynchronous exceptions [10]) are sent to the running threads, shown as dotted lines in Fig. 4. Our implementation is parametric to the precise algorithm that makes scheduler decisions, and in the following section we describe a number of such algorithms we have tried and evaluated.

A scheduler signal received by a thread will cause the update of the *local transactional state* of the thread, affecting the future execution of the thread. The local state of a thread is an object of the `TProcess` data-type:

```
data TProcess = TP {                data Alternative = A {
  expr :: Expression,                 tname :: TransactionID,
  ctx  :: Context,                    pr    :: TProcess }
  tr   :: [Alternative] }
```

The local state maintains the expression (`expr`) and evaluation context (`ctx`) currently interpreted by the thread and a list of *alternative* processes (represented by objects of the `Alternative` data-type). This list contains the continuations stored when the thread was embedded in transactions. The nesting of transactions in this list mirrors the transactional nesting in the global trie and is thus compatible with the transactional nesting of other expression threads. Let us go back to the example of page 6:

$$\begin{array}{ll}\llbracket (\mathbf{recv}\, c; \mathbf{commit}\, k)^{\mathtt{tid}_1} \parallel \llbracket (\mathbf{recv}\, d; \mathbf{commit}\, l)^{\mathtt{tid}_2} \rhd_l () \rrbracket & \\ \rhd_k (); \mathbf{atomic} \llbracket \mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l () \rrbracket \rrbracket & \parallel P^{\mathtt{tid}_P}\end{array}$$

where $P = \mathbf{send}\, d\, (); \mathbf{send}\, c\, ()$. When P is embedded in both k and l, the thread evaluating P will have the local state object

```
TP{expr = P, tr = [A{tname = l, pr = P}, A{tname = k, pr = P}]}
```

recording the fact that the thread running P is part of the l-transaction, which in turn is inside the k-transaction. If either of these transactions aborts then the thread will rollback to P, and the list of alternatives will be appropriately updated (the aborted transaction will be removed).

Once a transactional reconfiguration is performed by a thread, an acknowledgment is sent back to the gatherer, who, as we discussed, is responsible for updating the global transactional structure in the trie. This closes a cycle of transactional reconfigurations initiated from the process (by starting a new transaction or thread) or the scheduler (by issuing a commit, embed, or abort).

What we described so far is a simple architecture for an interpreter of TCML. Various improvements are possible (e.g., addressing the message bottleneck in the gatherer) but are beyond the scope of this paper. In the following section we discuss various policies for the scheduler which we then evaluate experimentally.

4 Transactional Scheduling Policies

Our goal here is to investigate schedulers that make decisions on transactional reconfiguration based only on runtime heuristics. We are currently working on more advanced schedulers, including schedulers that take advantage of static information extracted from the program, which we leave for future work.

An important consideration when designing a scheduler is *adequacy* [**?**, Chap. 13, Sec. 4]. For a given program, an adequate scheduler is able to produce all outcomes that the non-deterministic operational semantics can produce for that program. However, this does *not* mean that the scheduler should be able to produce all traces of the non-deterministic semantics. Many of these traces will simply abort and restart the same computations over and over again. Previous work on the behavioral theory of communicating transactions has shown that all program outcomes can be reached with traces that *never* restart a computation [13]. Thus a goals of our schedulers is to minimize re-computations by minimizing the number of aborts.

Moreover, as we discussed at the end of Sect. 2, many of the exponential number of embeddings can be avoided without altering the observable behavior of a program. This can be done by embedding a process inside a transaction only when this embedding is necessary to enable communication between the process and the transaction. We take advantage of this in a *communication-driven* scheduler we describe in this section.

Even after reducing the number of possible non-deterministic choices faced by the scheduler, in most cases we are still left with a multitude of alternative transactional reconfiguration options. Some of these are more likely to lead to efficient traces than other. However, to preserve adequacy we cannot exclude any of these options since the scheduler has no way to foresee their outcomes. In these cases we assign different, non-zero probabilities to the available choices, based on heuristics. This leads to measurable performance improvements, without violating adequacy. Of course some program outcomes might be more likely

to appear than others. This approach is trading measurable fairness for performance improvement.

However, the probabilistic approach is *theoretically fair*. Every finite trace leading to a program outcome has a non-zero probability. Diverging traces due to sequential reductions also have non-zero probability to occur. The only traces with zero probability are those in the reduction semantics that have an infinite number of non-deterministic reductions. Intuitively, these are unfair traces that abort and restart transactions *ad infinitum*, even if other options are possible.

Random Scheduler (R). The very first scheduler we consider is the random scheduler, whose policy is to simply, at each point, select one of all the non-deterministic choices with equal probability, without excluding any of these choices. With this scheduler any abort, embed, or commit actions are equally likely to happen. Although this naive scheduler is not particularly efficient, as one would expect, it is an obviously adequate and fair scheduler according to the discussion above. If a reduction transition is available infinitely often, scheduler R will eventually select it.

This scheduler leaves much room for improvement. Suppose that a transaction k is ready to commit:

$$[\![\, P \parallel \mathbf{co}\, k \rhd_k Q \,]\!]$$

Since R makes no distinction between the choices of committing and aborting k, it will often unnecessarily abort k. All processes embedded in this transaction will have to roll back and re-execute; if k was a transaction that restarts, the transaction will also re-execute. This results to a considerable performance penalty. Similarly, scheduler R might preemptively abort a long-running transaction that could have have committed given enough time and embeddings (for the purpose of communication).

Staged Scheduler (S). The staged scheduler partially addresses these issues by prioritizing its available choices. Whenever a transaction is ready to commit, scheduler S will always decide to send a commit signal to that transaction before aborting it or embedding another process in it. This does not violate adequacy; before continuing with the algorithm of S, let us examine the adequacy of prioritizing commits over other transactional actions with an example.

Example 1 *Consider the following program in which k is ready to commit.*

$$[\![\, P \parallel \boldsymbol{co}\, k \rhd_k Q \,]\!] \parallel R$$

If embedding R in k leads to a program outcome, then that outcome can also be reached after committing k from the residual $P \parallel R$.

Alternatively, a program outcome could be reachable by aborting k (from the process $Q \parallel R$). However, the $\boldsymbol{co}\, k$ *was spawned from one of the previous states of the program in the current trace. In that state, transaction k necessarily had the form: $[\![\, P' \parallel E[\boldsymbol{commit}\, k] \rhd_k Q \,]\!]$. In that state the abort of k was enabled. Therefore, the staged interpreter indeed allows a trace leading to the program state $Q \parallel R$ from which the outcome in question is reachable.* □

If no commit is possible for a transaction, the staged interpreter prioritizes embeds into that transaction over aborting the transaction. This is again an adequate decision because the transactions that can take an abort reduction before an embed step have an equivalent abort reduction after that step.

When no commit nor embed options are available for a transaction, the staged interpreter lets the transaction run with probability 0.95, giving more chances to make progress in the current trace, and with probability 0.05 it aborts it—these numbers have been fine-tuned with a number of experiments.

The benefit of the heuristic implemented in this scheduler is that it minimizes unnecessary aborts improving performance. Its drawback is that it does not abort transactions often, thus program outcomes reachable only from transactional alternatives are less likely to appear. Moreover, this scheduler does not avoid *unnecessary embeddings.*

Communication-Driven Scheduler (CD). To avoid spurious embeddings, scheduler CD improves over R by performing an embed transition only if it is *necessary* for an imminent communication. For example, in the following program state the embedding of the right-hand-side process into k will never be chosen.

$$\llbracket\, E[\mathbf{recv}\, c] \rhd_k Q \,\rrbracket \parallel ((); \mathbf{send}\, c\ v)$$

However, after that process reduces to an output, its embedding into k will be enabled. Because of the equivalence

$$\llbracket\, P \rhd_k Q \,\rrbracket \parallel R \equiv_{\mathrm{cxt}} \llbracket\, P \parallel R \rhd_k Q \parallel R \,\rrbracket$$

which we previously discussed, this scheduler is adequate.

For the implementation of this scheduler we augment the information stored in the trie data-structure (Sect. 3) with the channel which each thread is waiting to communicate on (if any).

As we will see in Sect. 5, this heuristic significantly boosts performance because it greatly reduces the exponential number of embedding choices.

Delayed-Aborts Scheduler (DA). The final scheduler we report is DA, which adds a minor improvement upon scheduler CD. This scheduler keeps a timer for each running transaction k in the transaction trie. This timer is reset whenever a communication or transactional operation happens inside k. Transaction k will only be considered for an abort when this timer expires. This strategy benefits long-running transactions that perform multiple communications before committing. The CD scheduler is obviously adequate because it only adds time delays.

5 Evaluation of the Interpreters

We now report the experimental evaluation of interpreters using the preceding Scheduling policies. The interpreters were compiled with GHC 7.0.3, and the experiments were performed on a Windows 7 machine with Intel® Core™i5-2520M 250 GHz processor and 8Gb of RAM. We run several versions of two programs:

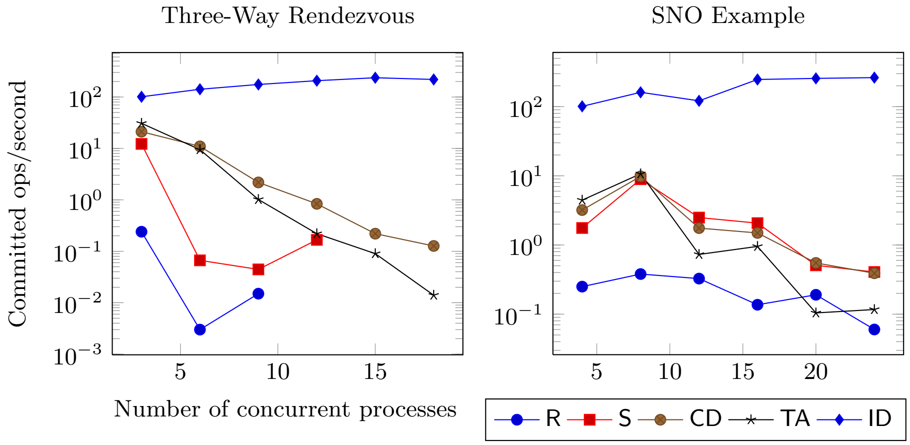

Fig. 5. Experimental Results.

1. The three-way rendezvous (3WR) in which a number of processes compete to synchronize on a channel with *two* other processes, forming groups of three which then exchange values. This is a standard example of multi-party agreement [11, 3, 5]. In the TCML implementation of this example each process nondeterministically chooses between being a *leader* or *follower* within a communicating transaction. If a leader and two followers communicate, they can all exchange values and commit; any other situation leads to deadlock and eventually to an abort of some of the transactions involved.
2. The SNO example of the introduction, as implemented in Sect. 2, with multiple instances of the Alice, Bob, Carol, and David processes.

To test the scalability of our schedulers, we tested a number of versions of the above programs, each with a different number of competing parallel processes. Each process in these programs continuously performs 3WR or SNO cycles and our interpreters are instrumented to measure the number of operations in a given time, from which we compute the *mean throughput* of successful 3WR or SNO operations. The results are shown in Fig. 5.

Each graph in the figure contains the mean throughput of operations (in logarithmic scale) as a function of the number of competing concurrent TCML processes. The graphs contain runs with each scheduler we discussed (random R, staged S, communication-driven, CD, and communication-driven with timed aborts TA) as well as with an *ideal* non-transactional program (ID). The ideal program in the case of 3WR is similar to the TCML, non-abstract implementation [11]. The ideal version of the SNO is running a simpler instance of the scenario, without any Carol processes—this instance has no deadlocks and therefore needs no error handling. Ideal programs give us a performance upper bound.

As predictable, the random scheduler (R)'s performance is the worst; in many cases R could not perform any operations in the window of measurements (30sec).

The other schedulers perform better than R by an order of magnitude. Even just prioritizing the transactional reconfiguration choices significantly cuts down

the exponential number of inefficient traces. However, none of the schedulers scale to programs with more processes; their performance deteriorates exponentially. In fact, when we go from the communication-driven (CD) to the timed-aborts (TA) scheduler we see worst throughput in larger process pools. This is because with many competing processes there is more possibility to enter a path to deadlock; in these cases the results suggest that it is better to abort early.

The upper bound in the performance, as shown by the throughput of ID is one order of magnitude above that of the best interpreter, when there are few concurrent processes, and (within the range of our experiments) two orders when there are many concurrent processes. The performance of ID is increasing with more processes due to better utilization of the processor cores.

It is clear that in order to achieve a pragmatic implementation of TCML we need to address the exponential nature in consensus scenarios as the ones we tested here. Our exploration of purely runtime heuristics shows that performance can be improved, but we need to turn to a different approach to close the gap between ideal ad-hoc implementations and abstract TCML implementations.

6 Conclusions and Future Work

Consensus is an often occurring problem in concurrent and distributed programming. The need for developing programming language support for consensus has already been identified in previous work on *transactional events* (TE) [3], *communicating memory transactions* (CMT) [9], *transactors* [4] and *cJoin* [1]. These approaches propose forms of restarting communicating transactions, similar to those described in Sect. 2. TE, CMT and Transactors can be used to implement the instance of the Saturday Night Out (SNO) example in this paper. TE extends CML events with a transactional sequencing operator; transactional communication is resolved at runtime by search threads which exhaustively explore all possibilities of synchronization, avoiding runtime aborts. CMT extends STM with asynchronous communication, maintaining a directed dependency graph mirroring communication between transactions; STM abort triggers cascading aborts to transactions that have received values from aborting transactions. Transactors extend actor semantics with fault-tolerance primitives, enabling the composition of systems with consistent distributed state via distributed checkpointing. The cJoin calculus extends the Join calculus with isolated transactions which can be merged. Merging and aborting are managed by the programmer, offering a manual alternative to TCML's nondeterministic transactional operations. It is unclear to us how to write a straightforward implementation of the SNO example in cJoin. Reference implementations have been developed for TE, CMT and cJoin. The discovery of efficient implementations for communicating transactions could be equally beneficial for all approaches. Stabilizers [14] add transactional support for fault-tolerance in the presence of transient faults but do not directly address concensus scenarios such as the SNO example.

This paper presented TCML, a simple functional language with build-in support for consensus via communicating transactions. This is a construct with a

robust behavioral theory supporting its use as a programming language abstraction for automatic error recovery [13, 2]. TCML has a simple operational semantics and can simplify the programming of advanced consensus scenarios; we introduced such an example (SNO) which has a natural encoding in TCML.

The usefulness of communicating transactions in real-world applications, however, depends on the invention of efficient implementations. This paper described the obstacles we need to overcome and our first results in a recently started project on developing such implementations. We gave a framework to develop and evaluate current and future runtime schedulers of communicating transactions and used it to examine schedulers which are based solely on runtime heuristics. We have found that some heuristics improve upon the performance of a naive randomized implementation but do not scale to programs with significant contention, where an exponential number of alternative computation paths lead to necessary rollbacks. It is clear that purely dynamic strategies do not lead to sustainable performance improvements.

In future work we intend to pursue a direction based on the extraction of information from the source code which will guide the language runtime. This information will include an abstract model of the communication behavior of processes that can be used to predict with high probability their future communication pattern. A promising approach to achieve this is the development of technology in type and effect systems and static analysis. Although the scheduling of communicating transactions is theoretically computationally expensive, realistic performance in many programming scenarios could be achievable.

References

[1] Bruni, R., Melgratti, H., Montanari, U.: cjoin: Join with communicating transactions. To appear in MSCS

[2] De Vries, E., Koutavas, V., Hennessy, M.: Liveness of communicating transactions. pp. 392–407. APLAS'10

[3] Donnelly, K., Fluet, M.: Transactional events. pp. 124–135. ICFP '06

[4] Field, J., Varela, C.A.: Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. POPL '05

[5] Harris, T., Marlow, S., Jones, S.P.L., Herlihy, M.: Composable memory transactions. Commun. ACM pp. 91–100 (2008)

[6] Herlihy, M., Shavit, N.: The art of multiprocessor programming. Kaufmann (2008)

[7] Jones, S.P.L., Gordon, A.D., Finne, S.: Concurrent haskell. POPL'96

[8] Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press (2008)

[9] Lesani, M., Palsberg, J.: Communicating memory transactions. PPoPP '11

[10] Marlow, S., Jones, S.P.L., Moran, A., Reppy, J.H.: Asynchronous exceptions in haskell. PLDI'01

[11] Reppy, J.H.: Concurrent programming in ML. Cambridge University Press (1999)

[12] Spaccasassi, C.: Transactional concurrent ml. Tech. Rep. TCD-CS-2013-01

[13] de Vries, E., Koutavas, V., Hennessy, M.: Communicating transactions. pp. 569–583. CONCUR'10

[14] Ziarek, L., Schatz, P., Jagannathan, S.: Stabilizers: a modular checkpointing abstraction for concurrent functional programs. ICFP '06